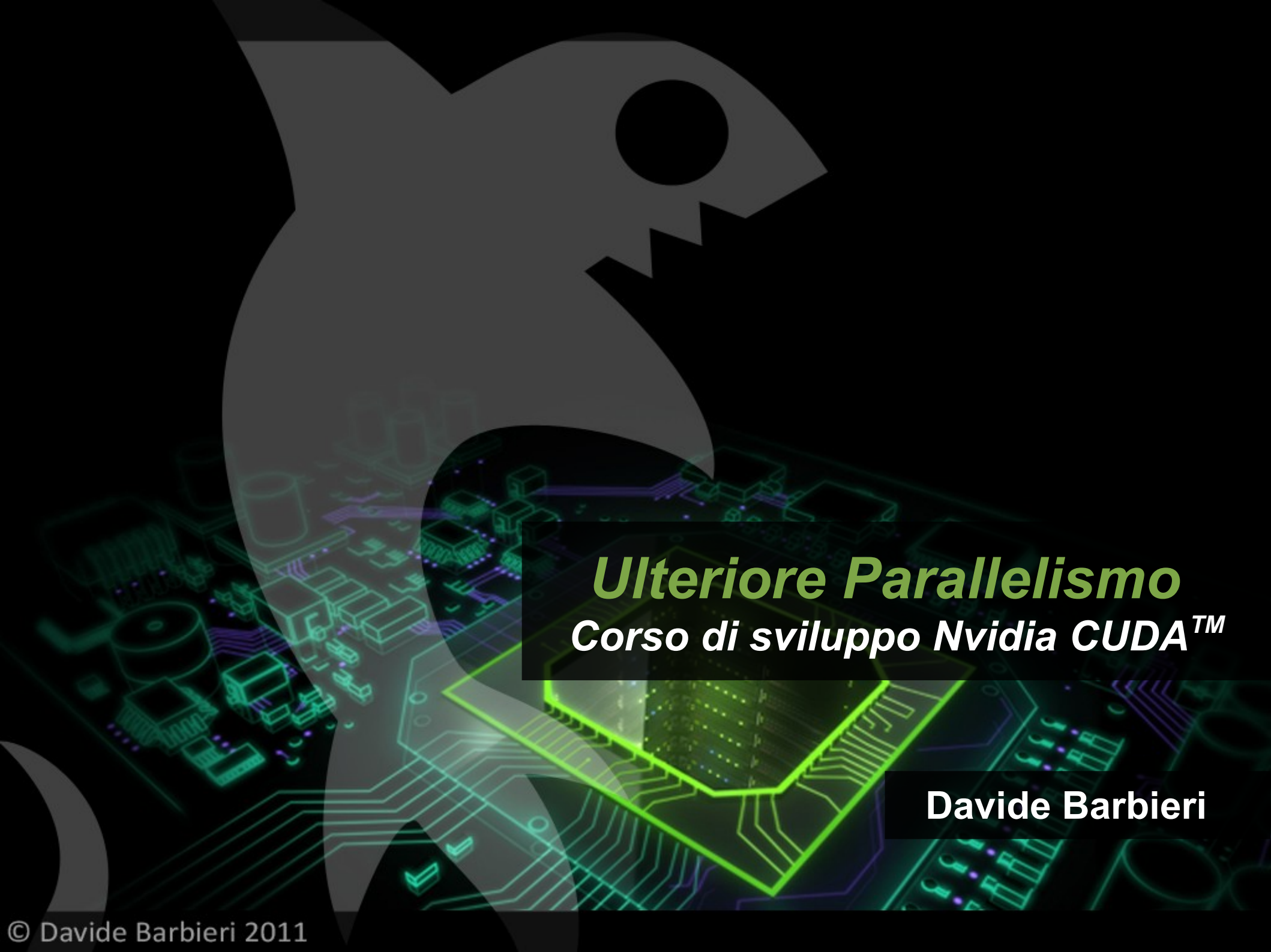


Ulteriore Parallelismo

Corso di sviluppo Nvidia CUDA™

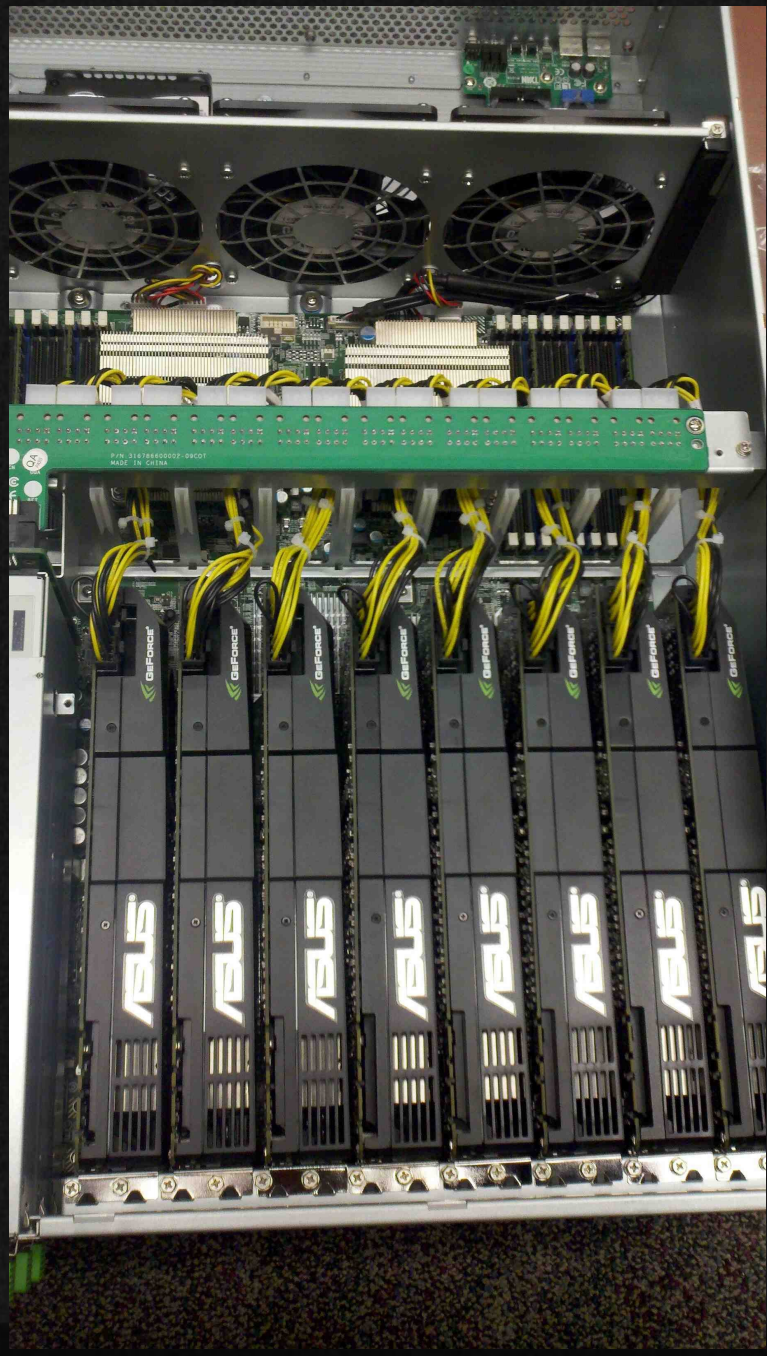
Davide Barbieri

Panoramica Lezione

- Soluzioni Multi-GPU
- Memoria *Page-Locked*
- *Unified Virtual Address*
- *CUDA Stream*
- Modalità *Zero-Copy*

CUDA Multi-GPU

- Un sistema può avere più GPU
  - Una scheda video a doppia GPU (**Nvidia GTX 590**)
  - Più schede video
    - Schede madri comuni fino a 4 slot PCI-Express
    - Soluzioni per Cluster (e.g. Microway)
      - 8 schede per nodo
  - Un cluster di nodi
    - PC connessi da Infiniband
    - Nvidia Tesla C2050/M2050/M2070/M2090



CUDA Multi-GPU

- Ogni GPU ha una memoria globale propria
- L'applicazione deve gestire gli spostamenti di memoria da GPU a GPU
  - Attraverso bus PCI-Express
    - Schede video a doppia GPU sono come due schede video separate
      - Ognuna utilizza 8 linee anziché 16

Contesto CPU-GPU

- Ad ogni device è associato un ***contesto*** *CPU-GPU*
- Mantiene traccia delle allocazioni Device richieste dal processo Host corrente
- Con la prima chiamata CUDA (cudaMalloc / cudaFree / kernel)
  - Inizializzazione di tutti i contesti (uno per device presente)
- In ogni momento, ogni host thread appartenente ad un processo
  - Avvia i comandi cudaMalloc/cudaFree e i kernel verso un contesto alla volta
- Enumerare le GPU attualmente presenti
  - cudaGetDeviceCount()
  - cudaGetDeviceProperties()
- Selezionare il contesto corrente
  - cudaSetDevice()

Contesto CPU-GPU

- Ogni contesto viene distrutto all'uscita del processo
  - Liberando memoria allocata

- *cudaDeviceReset()* permette di distruggere il contesto del Device correntemente selezionato

Memoria Page-Locked (Host)

- ***Page-Locked* o *Pinned Memory***
  - Si chiede all'OS di sottrarre quegli indirizzi al *paging*
  - Permette gestione asincrona della memoria
    - *cudaMemcpyAsync()*
  - Non abusarne (rallenta tutto il sistema)

- ***Write-Combining Memory***
  - Disabilita la gerarchia di *cache (Host)*
  - Proibitivo leggere da CPU
  - Veloce scrivere da CPU
  - Utilizzare solo per scrivere dati da passare alla GPU
  - Fino al 40% di velocità in più

- ***Mapped Memory***
  - *Mapping di memoria Host sullo spazio di indirizzi Device*
  - Caricamenti memoria host direttamente da Device
  - *(modalità **Zero-Copy**)*

- Per allocare Pinned Memory
  - ***cudaHostAlloc()/cudaFreeHost()***
  - ***cudaHostRegister()/cudaHostUnregister()*** per rendere *pinned* aree di memoria allocate con malloc()
  - Per condividerla tra contesti diversi
    - flag *cudaHostAllocPortable* / *cudaHostRegisterPortable*

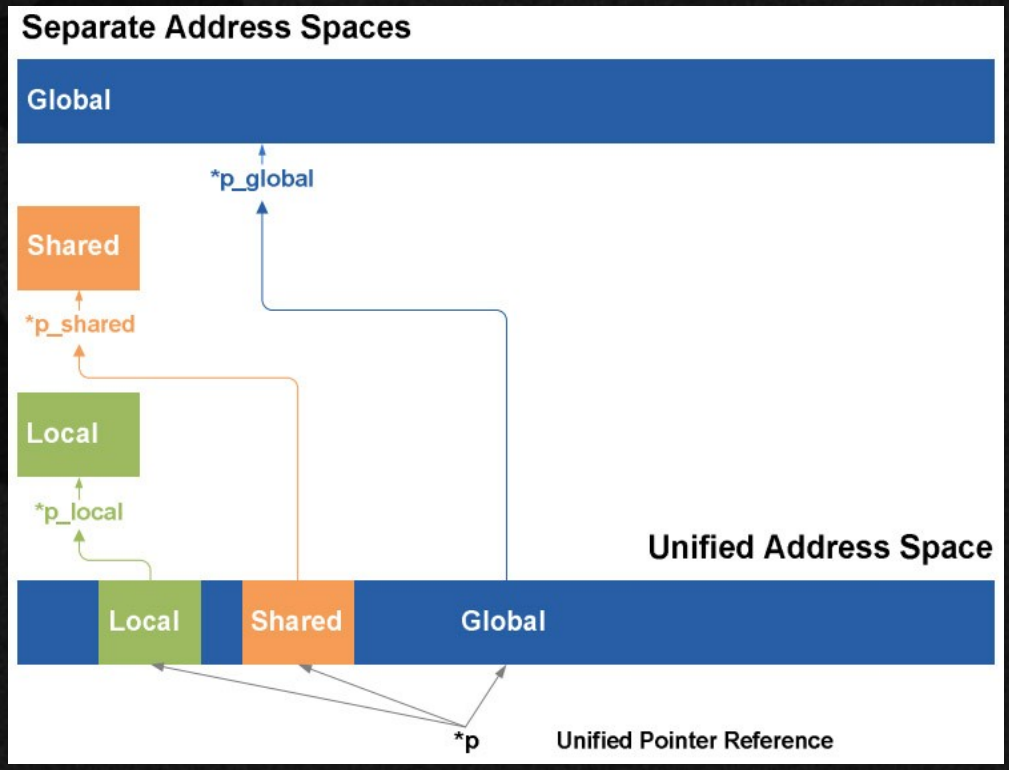
- Per allocare Write-Combining Memory
  - *cudaHostAllocWriteCombined* → cudaHostAlloc()

- Per mappare memoria host sul device
  - *cudaHostAllocMapped* → cudaHostAlloc()
  - *cudaHostRegisterMapped* → cudaHostRegister()

  - Una volta mappata, l'area host avrà in genere 2 indirizzi:
    - Indirizzo Host (dell'area effettivamente allocata)
    - Indirizzo Device (che mappa quello Host)
    - *cudaHostGetDevicePointer()*

Unified Virtual Address

- L'hardware Fermi è in grado di mappare sul suo spazio di indirizzamento a 64 bit:
  - Memoria Host
  - Memoria Condivisa
  - Memoria Globale
  - Memoria Locale
    - Stack frame
  - Memoria Globale di altri dispositivi



→ Gestione puntatori più flessibile
→ Supporto puntatori a funzione

- Per applicazioni 64-bit che utilizzano Fermi:
  - Windows (opportunamente configurato) o Linux
  - Spazio di indirizzamento Host e Device sono condivisi

- Fermi con UVA abilitato
  - La memoria host è automaticamente mappata su memoria Device
  - *I kernel possono utilizzare direttamente l'indirizzo dato da **cudaHostAlloc()**, senza dover utilizzare **cudaHostGetDevicePointer()***
    - **I due indirizzi (Host e Device) coincidono**

Perché mappare memoria host?

- Quando un kernel lavora su dati prodotti dall'Host sempre nuovi
- Ad esempio, che arrivano dalla **rete**
  - Nascondiamo la latenza memoria Host ↔ Device
- Con cudaMemcpy, necessario:
  - Allocare un buffer in memoria device
  - Attendere il completamento della copia prima di lanciare il kernel
- Con mapped memory:
  - L'allocazione non è necessaria
  - Le transazioni di memoria host → device solo quando richieste dal kernel (come una sorta di pipeline)

    → minor latenza
- Questa modalità è detta *Zero Copy*
  - non ho bisogno di stream per nascondere la latenza host ↔ device
  - Sulle GPU **integrate** memoria Device e Host coincidono
    - Utilizzate questa modalità per evitare copie inutili

CUDA Stream

- ***Stream***:
  - come una coda **FIFO** di comandi
    - Transazioni di memoria
      Host ↔ **Device** o **Device** ↔ **Device**
    - Chiamate a kernel

- Lo stream è associato ad un **solo** device
  - Ogni device ha implicitamente associato uno stream di default
- All'interno dello stesso Stream chiamate sincrone

- Utilità Stream:
  - Sovrapporre memcpy ad esecuzioni kernel di stream diversi
  - Più kernel concorrenti su Fermi (uno per Stream)
  - Multi-GPU
    - Ogni stream gestisce un device

CUDA Stream

- Funzioni runtime per gestire gli stream
- Creazione/Distruzione stream:

```
cudaStream_t stream0;
cudaStreamCreate(&stream0);
cudaStreamDestroy(stream0);
```

- Esecuzione su stream:

```
kernel<<<grid,block,shareMem,stream0>>>();
```

- Copia associata a stream:

```
cudaMemcpyAsync(dst,src,size,kind,stream0);
```

- Sincronizzazione con l'host:
  (l'host aspetta il completamento di tutti i comandi in coda Stream)

```
cudaStreamSynchronize(stream0);
```

Eventi CUDA

- Oltre ai comandi finora studiati, possiamo inserire negli Stream dei segnaposto chiamati ***Eventi***
  - Utili per monitorare i progressi di uno stream

- Creazione/Distruzione:
  cudaEvent_t event0;
  cudaEventCreate(&event0);
  cudaEventDestroy(event0);

- Inserimento evento in uno stream:
  cudaEventRecord(event0, stream0);

- Wait lato host di un evento:
  cudaEventSynchronize(event0);

Eventi CUDA

- Wait dello stream stream0 di un evento:
  cudaStreamWaitEvent(stream0,event0)

  - Ovvero carico sulla coda dello stream
    un comando di wait di un evento (di un altro stream)

  - Una volta che quello stream è giunto alla wait si blocca

    - Riprende solo quando lo stream associato all'evento
      ha completato tutto fino a quell'evento
      (Se lo stream è 0, la wait è direzionata a tutti gli stream)

  - Utile per sincronizzare stream di dispositivi diversi

- Timing (in millisecondi) tra due eventi (accurato):
  float elapsedTime;
  cudaEventElapsedTime(&elapsedTime, event0, event1);

Esempio Multi GPU (Host thread singolo)

```
for (int i=0; i<numIterazioni; ++i)
{
 cudaSetDevice(0);
 kernel<<<grid, block, 0, stream0>>>(dataA);
 cudaEventRecord(event0, stream0);

 cudaSetDevice(1);
 kernel<<<grid, block, 0, stream1>>>(dataB);
 cudaEventRecord(event1, stream1);

 cudaStreamWaitEvent(stream0, event1);
 cudaStreamWaitEvent(stream1, event0);

 // copio i dati di confine tra schede
 cudaMemcpyAsync(..., stream0);
 cudaMemcpyAsync(..., stream1);
 cudaEventRecord(event2, stream0);
 cudaEventRecord(event3, stream1);

 // Synchronize the devices with each other
 cudaStreamWaitEvent(stream0, event3);
 cudaStreamWaitEvent(stream1, event2);
}
```

Stream 0 (GPU 0)	Stream 1 (GPU 1)
Kernel(dataA)	
Event0	
	Kernel(dataB)
	Event1
----------	**----------**
Memcpy(confineA)	
Event2	**Memcpy(confineB)**
	Event3
----------	**----------**

Esempio Multi GPU (Host thread multipli)

Thread 0

```
cudaSetDevice(0);

for (int i=0; i<numIterazioni; ++i)
{

 kernel<<<...,stream0>>>(dataA);

 cudaDeviceSynchronize();
 pthread_barrier_wait(b)

 // copio i dati di confine tra schede
 cudaMemcpyAsync(...,stream0);

 cudaDeviceSynchronize();

 pthread_barrier_wait(b);
}
```

Thread 1

```
cudaSetDevice(1);

for (int i=0; i<numIterazioni; ++i)
{

 kernel<<<...,stream1>>>(dataB);

 cudaDeviceSynchronize();
 pthread_barrier_wait(b)

 // copio i dati di confine tra schede
 cudaMemcpyAsync(..., stream1);

 cudaDeviceSynchronize();

 pthread_barrier_wait(b);
}
```

Accesso Peer to Peer

- Accessi Memoria **peer-to-peer** (da GPU a GPU)
  - Copie dirette da GPU a GPU, anziché utilizzare un buffer host intermedio

- Un kernel su Fermi è capace di dereferenziare un puntatore che punta memoria di un'altra GPU
  - Supportato solo da soluzioni senza uscita video
  - Solo se entrambe le GPU supportano UVA
  - *cudaDeviceCanAccessPeer()*
  - *cudaDeviceEnablePeerAccess(peer, flags)*

- Con UVA, è possibile copiare da device a device tramite semplici cuda-Memcpy

- Se UVA non è supportato:
  - *cudaMemcpyPeer (dst, dstDevice, src, srcDevice, count)*